LU02b - Konzept des data hiding

Unter **data hiding** (Datenkapselung) versteht man den Zugriff auf die Attribute eines Objektes nur mittels Methoden und nie im Direktzugriff https://de.wikipedia.org/wiki/Datenkapselung_(Programmierung).

Im UML-Diagramm werden daher die Attribute mit dem Modifikator «**private**», hier ein **-** Zeichen versehen.

Abb. private-Modifikator in UML

Was bedeutet nun aber data hiding praktisch betrachtet?

Wir nehmen hier wieder unser Beispiel der Türe. Gemäss UML-Diagramm sind alle Attribute `private` deklariert. Ein Zugriff in der Art

```
some_door = Door()
some_door.door_is_open = True
```

ist somit nicht erlaubt. Wäre dem nicht so, könnte z.B. auch im Zustand `verriegelt` die Türe geöffnet werden, was aber gar nicht geht. Viel mehr wird in der Methode `lock_the_door` sichergestellt, dass eben die Bedingungen gemäss Zustandsdiagramm eingehalten werden.

```
def lock_the_door(self):
    """
    Methode für das verriegeln der Türe.
    Das ist nur möglich, wenn die Türe nicht offen ist.
    Für das verriegeln ist aber das Türschloss zuständig. Es weiss wie das
geht.
    """
    if self._door_is_open == False:
       self._door_is_locked = self._the_door_lock.lock()
```

Der Benutzer der Klasse `Door` ist also vollständig von der Art der Implementation befreit und muss sich um keinerlei Zusicherungen kümmern. Es genügt, wenn er die passenden Methode aufruft. Hier also

```
some_door.lock_the_door()
```

data hiding bei Python

Attribute

Attribute die im Klassendiagramm als `private` deklariert sind, müssen in Python im Konstruktor initialisiert werden. Dies erfolgt durch folgende Schreibweise

```
self._attributname = initial_wert
```

Im Gegensatz zu anderen Programmiersprachen gibt es in Python keine echten **private** Attribute. Mit der Schreibweise `_attributname` zeigen wir an, dass dieses Attribut nicht von ausserhalb der Klasse manipuliert werden soll. Es wird aber nicht durch den Python-Interpreter verhindert, dass trotzdem auf solche Attribute zugegriffen wird.

Property und Setter

Durch den Einsatz von `@property` und `@`*`ATTRIBUT`*`.setter` können wir das Prinzip des Data Hidings unterstützen. Als Beispiel betrachten wir das Attribut `color` der Klasse `Door`.

```
 1. class Door:
 2.     def __init__(self, ref2door_lock, base_color):
 3.         self.color = base_color
 4.
 5.     @property
 6.     def color(self):
 7.         """
 8.         getter-Methode für die Eigenschaft color
 9.         :return: die Farbe des Objekts
10.         """
11.         return self._color
12.
13.     @color.setter
14.     def color(self, new_color):
15.         """
16.         setter-Methode für die Eigenschaft color
17.         :param new_color:
18.         """
19.         self._color = new_color
20.
21. if __name__ == '__main__':
22.     some_door = Door(None, 'blue')
23.     print(f'The color is {some_door.color}')
```

```
24.     some_door.color = 'red'
25.     print(f'The color is {some_door.color}')
```

Auf den ersten Blick scheint es, als würde das Programm in Zeile 24 direkt auf das Attribut `color` zugreifen. Tatsächlich erkennt Python die Decorators `@property` und `@color.setter` und leitet die Zugriffe auf die entsprechenden Methoden um.
